International Journal of Enhanced Research in Science Technology & Engineering, ISSN: 2319-7463
Vol. 3 Issue 1, January-2014, pp: (12-23), Impact Factor: 1.252, Available online at: www.erpublications.com

Optimizing Packet Filtering Libraries for Scada System Security

Tanmay Sinha[1], Tanisha Parul[2]
[1]Department of Computer Science, Vellore Institute of Technology, Chennai, India
[2]Department of Computer Science, Jaypee Institute of Information Technology, Noida, India

Abstract: Supervisory Control and Data Acquisition (Scada) systems have been traditionally used since a long time to control the critical infrastructure of many countries. The amount of complexity in Scada networks coupled with lack of security has caused an urgency to upgrade existing systems to withstand hostile attacks. In our work, we dissect these large Scada networks at the protocol level on which they operate, and further dive into the packet filtering libraries that comprise the fundamental skeleton of such Scada systems. Incidentally, we explore at a micro level whether, and how, the design of such libraries reflect potential problems in Scada networks governing the operation of large geographical areas. While previous studies show that protocol level security mechanisms reflect the robustness of communication networks to some extent, this is certainly not the case for all such networks (for example, thinking about massively scalable Scada networks) and thus, it is legitimate to ask whether flaws may also be revealed and overcome through gaining deeper insights on how packet filtering libraries function. In the current study, we question first whether such results on Scada system security be extended to more ecologically valid situations and secondly, whether we can find relationships in the solutions that we draw extensively from the domains of protocol design, network programming and operating system security.

Keywords: Security, Libipq, Libpcap, Packet Filter, Performance Tuning, Scada.

1. Introduction

Scada is an acronym for Supervisory Control and Data Acquisition. Scada systems are used to monitor and control a plant or equipment in industries such as telecommunications, water and waste control, energy, oil and gas refining and transportation. Typically, these systems encompass the transfer of data between a Scada central Master Terminal Unit (Mtu) and a number of Remote Terminal Units (Rtu's) and/or Programmable Logic Controllers (Plc's) using Human Machine Interfaces (Hmi's) and Intelligent Electronic Devices (Ied's) [1]. Such Scada systems are built using public or proprietary communication protocols which are used for communicating between an Mtu and one or more Rtu's. Scada protocols provide transmission specifications to interconnect substation computers, Rtu's, Ied's, and the master station. Scada security is very essential in today's world, because a nation's critical infrastructure (such as power grid, water or oil supply, chemical processing plants etc) may be damaged because of breaches in security. Hence, understanding the fundamental protocols on which Scada systems operate, is a prerequisite for developing solutions to protect them. This will enable the introduction of novel security mechanisms at both hardware and software levels, with varying degrees of granularity depending on requirements. In this work, we are interested in demonstrating how basic packet filtering libraries, after optimization and removing their current loopholes, be applied to Scada systems in a computationally feasible manner.

Thus, we adopt an incremental approach, wherein we introduce some of the most important protocols used for Scada systems in Section 2, the motivation for studying Scada system security in Section 3, adaptive packet filtering library optimization in Section 4, a generalized solution for hardware based dynamic filtering in Section 5, along with the analysis of high speed packet capturing in Scada systems in Section 6. We conclude with future research directions and conclusion in Section 7 and 8 respectively.

2. Scada System Protocols

A. DNP3

The objectives of any communication protocol are minimizing bandwidth, ensuring reliable data transfer, providing features like freeze operations, timestamps, quality flags and preventing unauthorized use of data. The most common protocols used for Scada communication are: Iec (International Electrotechnical Commission) 60870-5-101 [2], Dnp3 (Distributed Network Protocol version 3.0) [3] [4], and Modbus. Based on public distribution, greater functionality and extensive use, Dnp3 plays a major market role around the world. It is designed to optimize the transmission of data acquisition information and control commands from master (Mtu) to the outstation (Rtu) and follows a very modular layering approach for data flow over TCP/IP stack. Outstation computers gather data from physical sensors and actuators for transmission to the master. They may do so, either when they are polled by the Mtu, or during an unsolicited response (similar to a vectored interrupt). Data may binary, analog, counter values or file related. Master station gives output values (command to perform certain action) to the Rtu. It must always keep its database updated.

The Dnp3 protocol stack consists of: (i) Application layer- which is associated with processing complete message data for requesting, or responding to requests. Application layer messages are broken into fragments with each fragment size usually a maximum of 2048 bytes. It is the responsibility of the application layer to ensure the fragments are properly sequenced. (ii) Transport Layer- which is responsible for breaking long application layer messages into smaller segments sized for the link layer to transmit, and, when receiving, to reassemble frames into longer application layer messages. (iii) Data Link Layer- which handles the transmission and reception of data frames across the physical layer. Data link layer frames contain two cheksum bytes every 16 bytes. It makes the physical link reliable (by error detection and duplicate frame detection), while sending and receiving packets called frames. Dnp3 Objects are each and every instance of an encoded information element, defined by a unique group and variation within the message. They may be static objects (depend on current data) or event objects (triggered by significant events like state change, exceeding threshold limits etc). Objects are in-turn divided into groups (analog/digital/counter/file operations) and each group has variations (16/32 bit integer or floating point value with flags, timestamps, prefixes or vice versa). The application and transport headers have the sequence number for the fragment/segment, and also the first and final bits to indicate the beginning and end. This helps in reassembly of the frames at the initiating or responding side in a Scada system.

The benefits of using the internet technology to carry Scada communications come at the cost of compromised security, since the data over the internet can be an easy target for an attack. To make the situation more challenging, Dnp3 has no built-in security feature considerations such as message encryption. Various threats that Dnp3 faces include eavesdropping, man-in-the-middle attack (in which a malicious hacker not only listens to the messages between two unsuspecting parties but can also modify, delete, and replay the messages), spoof and replay (an attack that attempts to trick the system by retransmitting a legitimate message) [5].

B. DNPSEC

The motivation to study DnpSec Protocol [6] comes directly from the above stated problems in Dnp3 Protocol. The main goal is to address the threats related to confidentiality, integrity, and authenticity in the Scada Systems using the Dnp3, with a minimum performance impact on the communication link, and without requiring modification to the much more expensive master station and substation units along with the applications supporting them. There are two main components of the DnpSec. First is the new DnpSec structure to construct the frame and transfer data in secure mode between the master and the slave. Second is symmetric key exchange established during the installation and connection setup between the master and the slave.

Some of the crucial roles DnpSec is supposed to play include verifying the frame origin, assuring that the frame sent is the frame received, assuring that the network headers have not changed since the frame was sent and giving anti-replay protection, encrypting frames to protect against eavesdropping and hiding frame source by applying encryption methods. So, to achieve this, appropriate modifications are done at the Data Link Layer of Dnp3 Protocol. In simplistic terms, a counter value increases by one for each message sent by the master. If the limit of $2^{32} - 1$ is reached, the master terminates the session key and sends a new key to the slave encrypted with the previous session key. The attacker cannot recover the session key without the previous session key. This kind of functionality guarantees that the master and the slave continue establishing a new session key even if the connection is always open. 20 bytes of authentication data (with an Integrity Check Value) is added to Dnp3 Data Link frame using algorithms like Md5, Sha1 etc. Delivery time in DnpSec equals the sum of the encryption speed, the decryption speed, encryption key set up, decryption key set up and the transmission time.

However, this protocol also has some loopholes. There are thousands of existing networks that use the legacy Scada protocols. These networks cannot be replaced overnight. So, there is a need to make changes in the existing protocols and in such a way that existing equipments and infrastructure are least affected. These problems are proposed to be improved by a new protocol, called FlexiDnp3.

C. FLEXIDNP3

This protocol works by use of an embedded device called Bump-In-The-Wire (Bitw) [7], which seems an ideal solution to the above stated problem with DnpSec. Since, the Bitw is external to the Scada device, it doesn't require making changes in existing Rtu's and Mtu's. Figure 1 shows the placement of this device. The necessity to introduce FlexiDnp3 is that an outstation can use this more secure protocol to unambiguously determine that it is communicating with an Mtu which is authorized to access the services of the outstation. It aims to provide security to the Dnp3 protocol without increasing the message size or deleting any data from the packet header. There is some existing body of literature on novel Bitw solutions for retrofitting security to time-critical communications in serial-based Scada systems [18].

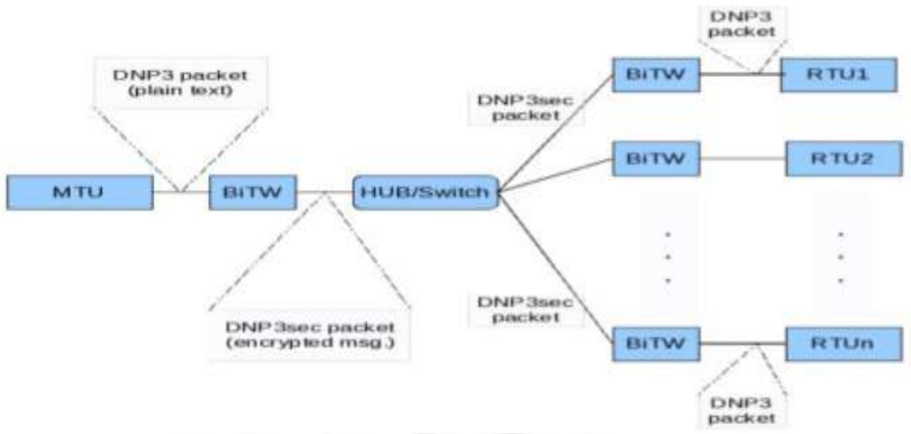

Figure 1. Use of BITW to control communication between Master and Remote Terminal Units in FlexiDnp3

In the FlexiDnp3 protocol, the modifications made to DnpSec protocol are: (i) Size of the DnpSec packet is fixed at 292 irrespective of the size of the Dnp3 packet. The payload, if lesser than that, is padded with dummy data so that payload size can be made 256 bytes. Along with DnpSec header and the authentication data, it comes to 292 (ii) DnpSec packet is identified from Dnp3 and other packets on network by finding sync bytes 0x0564 at byte position 8-9 of the DnpSec packet. Consequently only payload data (256 bytes) is encrypted instead of encrypting payload and original link header. This scheme provides the same confidentiality level as the original DnpSec scheme. The original DnpSec protocol has key sequence Number (Ksn). When Ksn reaches maximum and is re-cycled to 0, the Mtu sets SK (session key) bit and sends the new session key. Instead of this arrangement, the SK bits are reserved and instead SecKeyD key distribution protocol [7] is used to negotiate the key and freshness number between Bitw's of Mtu and Rtu. And, this negotiation of keys can be done after a configurable number of data exchanges.

For the implementation, each Mtu and Rtu is connected to an external Bitw. Seed-keys are stored in all Bitw's (mtu and rtu side). After every 'n' message exchanges between a Rtu and Mtu, we exchange a symmetric key between Rtu's Bitw and Mtu's Bitw. Aes (Advanced Encryption Standard) algorithm is used for encryption and decryption of the messages containing keys exchanged between Rtu's and Mtu, because it provides more security than Blowfish for the same number of bits. Bitw captures the Dnp3 packet coming from Mtu and converts it to DnpSec packet after encrypting the payload using the secret symmetric key. It not only encapsulates Dnp3 packet within DnpSec and adds Authentication Data for integrity check (Sha1 hashing algorithm used), but also adjusts the TCP/IP cheksum and payload length to reflect these changes.

Packet filtering libraries are used to take these captured packets at the kernel level and make them available to the user-level application for processing. So, this brings in the need to study and apply carefully selected security mechanisms to packet filtering kernel libraries, which can be interfaced with the operating System to enable crafting and filtering of FlexiDnp3 packets.

3. Motivation

Before analyzing loopholes and formulating optimization solutions for packet filtering kernel libraries, we must understand the different ways in which security breaches are practically possible in Scada systems. Jain et al. [8] outline a comprehensive view of the same. Therefore, in this section, we outline how can breaches be physically realized and motivate the real time implications of developing robust security mechanisms for Scada networks. The general principles of security can be clubbed in groups of secure and bidirectional (two pass) address authentication, backwards tolerance (a secure device must be able to detect that the non-secure device does not support the authentication mechanism and the two devices must be able to continue to exchange information), forward secrecy (if a session key is compromised, this mechanism only puts data from that particular session at risk, and does not permit an attacker to authenticate data in future sessions) and upgradability (easy change of algorithms, key lengths, and other security parameters to deal with future requirements).

Talking specifically about Scada systems, an intruder can use protocol analyzer tools such as Wireshark to intercept the Dnp3 frames. As a result, he grabs unencrypted (plaintext) frames from a Dnp3 Scada system network application and the address of the source and destination systems. Attacks could take the form of shutting off the Mtu software, shutting down the Mtu computer, or making the Rtu stop functioning. Unsolicited message (alarming) generation for event reporting is configurable by the Master Station through the usage of the configuration functions in the application function code. The intruder can even intercept the connection between master and outstation (no authentication required).

He can very well masquerade as sender to the receiver. So, he may first disable unsolicited alarming messages that the outstations can send to the master by sending some specified function codes, so that they can't report any abnormal behavior. Then, he can make modifications as he wishes to the Dnp3 stack and broadcast the wrong packets to all states (by using address 65535). This can cause massive physical failures. At this stage, Mtu's may give wrong readings. Therefore, authentication is required both at master and client's end. Thus, the proposed protocols, DnpSec and FlexiDnp3 are an attempt to provide an end to end security to these Scada networks. Analyzing important security aspects are useful in real time for a deeper analysis of connection attempts, detecting modification in lots of files/logs/binaries generated in Scada networking, understand packet crafting and Sniffing, and improving network performance of TCP/IP (because Dnp3 protocol works on this stack only).

4. Packet Filtering Library Optimization

Packet filters are essential to build fundamental network services ranging from traffic monitoring to network engineering and intrusion detection. The underlying mechanism is made more comprehensible in figure 2. In recent years, with constantly increasing network speed and increasing protocol complexity, packet filters have been facing tough challenges posed by more dynamic filtering tasks and faster filtering requirements. However, existing packet filter systems have not yet fully addressed these challenges in an efficient and secure manner. The Linux kernel includes tools for performing both shallow header-based filtering and deep filtering (quick access to the full contents of network packets and the ability to modify those contents while the packet in transit). Our objective in this section is to clearly portray the problems with the currently used approaches in packet filtering, how they can be resolved and the whole process can be optimized for Scada systems. In this perspective, we discuss loopholes and improvements for two basic packet filtering libraries: libipq [9] and libpcap [10].

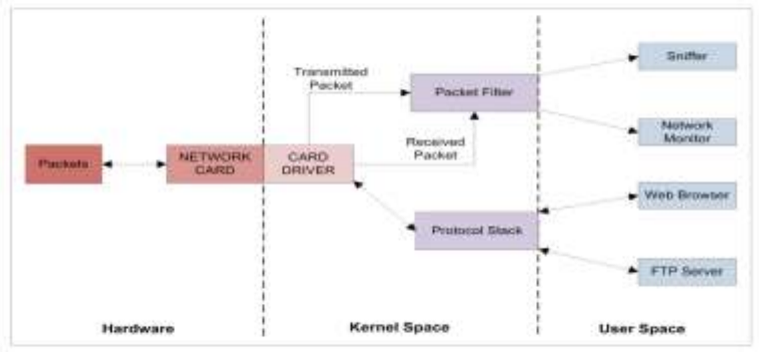

Figure 2. Basic Packet filtering mechanism

A. LIBIPQ

The GNU/Linux family of operating systems includes a packet filtering framework known as netfilter. One of the most important parts of netfilter is iptables, a command-line utility for viewing, adding, removing, and modifying firewall rules dynamically. But, iptables does not examine the actual packet data (the payload). There are times, however, when it would be useful to control the flow of packets based on their contents. Then, we can't use iptables alone. Deep packet filtering in Linux is provided by ip_queue kernel module. This module allows the kernel to delay or queue certain packets that match the appropriate rules, for further processing. To the programmer, this is presented as the Libipq application programming interface, as described in figure 3.

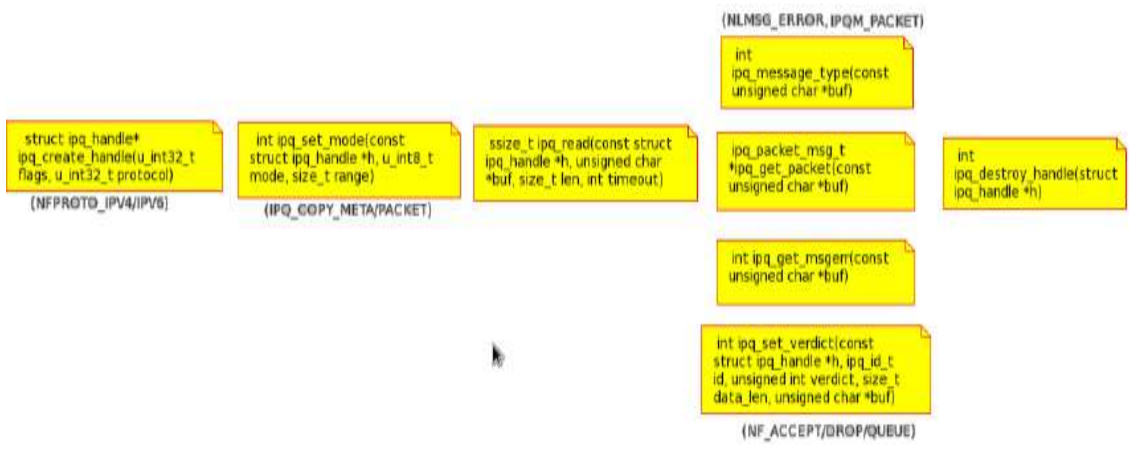
(NLMSG_ERROR, IPQM_PACKET)
int ipq_message_type(const unsigned char *buf)
struct ipq_handle* ipq_create_handle(u_int32_t flags, u_int32_t protocol)
(NFPROTO_IPV4/IPV6)
int ipq_set_mode(const struct ipq_handle *h, u_int8_t mode, size_t range)
(IPQ_COPY_META/PACKET)
ssize_t ipq_read(const struct ipq_handle *h, unsigned char *buf, size_t len, int timeout)
ipq_packet_msg_t *ipq_get_packet(const unsigned char *buf)
int ipq_destroy_handle(struct ipq_handle *h)
int ipq_get_msgerr(const unsigned char *buf)
int ipq_set_verdict(const struct ipq_handle *h, ipq_id_t id, unsigned int verdict, size_t data_len, unsigned char *buf)
(NF_ACCEPT/DROP/QUEUE)

Figure 3. Summary of different function calls that characterize Libipq library

Because this module is very application specific, therefore it's not apt to include this module in the kernel. Therefore the first option is building many dynamically loadable modules for the purpose. But, this is bound to have security and stability issues (difficult to protect operating system from errors in filtering software). So, deep filtering has to be carried out at the user level only. Using the netfilter ip_queue kernel module and the accompanying libipq C library, we can develop a user packet filtering daemon that partners with the kernel to provide deep packet filtering. Such a mechanism will basically work by the ip_queue module interfacing with the kernel and introducing new functionality into iptables. Rather than accepting or rejecting a packet that is matched by one of the rules in the kernel's firewall table, a rule may be cause a packet to be queued for further processing. This is the catching point here. The queued packet will be sent to a daemon running on the system, processed, and then accepted or modified or rejected. Packets can be received from the kernel by the processing daemon via a special socket connection using a protocol called netlink firewall, part of the pf_netlink protocol family. The data structure involved in the whole operation is "struct sk_buff", which stores a single datagram fragment, in which the data field in the structure is a pointer to the packet payload. When a datagram is fragmented, it is stored in the kernel as a linked list of sk_buff structures. However, struct sk_buff is normally destroyed when the packet is sent on for delivery, dropped, or otherwise done with. So, when ip_queue selects packets for deep filtering, it must make extra provisions for storing those packets (new data structure used is struct ipq_queue_entry). Based on the explanations from Stevens et al. [19], we can summarize the whole operation of this library as:

1. When a packet arrives on a network interface, the kernel scans its iptables firewall rule set for the first rule that matches the packet. That rule may instruct the kernel to queue the packet.
2. The kernel then crafts a message to send to the queue processing daemon. This message contains the packet's header information along with the contents of the packet itself.
3. The message is sent to the daemon via the special socket connection.
4. The daemon reads the message, extracts and examines the packet information, and determines a verdict for the queued packet.
5. Finally, the daemon sends a socket message back to the kernel indicating its verdict.

Here, the biggest problem is that the original packet has been removed from the stream of packets passing through the network host, but it remains stored in kernel memory in the ip_queue module's list of queued packets. Because the entire contents of the packet are copied into a new area of kernel memory before being pushed onto the netlink socket, this leads to slowdown for larger networks such as Scada systems.

Proposed Solution: It's better, if we do not copy the contents of the incoming packet into a new data structure to be sent for processing via the socket connection. Instead, we only supply the incoming packet's address in kernel memory. The user program can then directly accesses the packet's contents in kernel memory. Direct access to kernel memory is provided by the /dev/kmem pseudo-device. It's a character device (file) that can be accessed via open( ), read( ), write( ) system calls. The file kmem is the same as mem, except that the kernel virtual memory rather than physical memory is accessed. It is typically created by the following command: # mknod -m 640 /dev/kmem c 1 2 and #chown root:kmem /dev/kmem. The file will have permissions 640, major number as 1, minor number as 2, with root being the owner of the character file. By passing an address to the lseek64( ) system call, we can access any portion of kernel memory. The command which can be used is #off64_t lseek64 (int fd, off64_t offset, int whence). This will reposition the offset of the open file associated with the file descriptor 'fd' to 'offset' bytes relative to the start, current position, or end of the file, when 'whence' has the value Seek_Set, Seek_Cur, or Seek_End.

However, there are certain risks with this solution too. Firstly, there is reliance on the availability of the /dev/kmem pseudo-device. Secondly, security vulnerability could be an issue because, accessing kernel memory may hamper the normal working of the operating system. Domain switching or privilege escalation may violate the Principle of Least Privilege [11], an important aspect of protection. Thirdly, getting the appropriate address of the kernel virtual memory of where to create the character file for packet queuing, also poses risk of interfering with high priority kernel related processes that are executing.

B. LIBPCAP

Traditional packet-filtering firewalls control network traffic based on pre-defined rules. Dynamic filtering tasks refer to on-line packet filtering procedures in which filtering criteria frequently change over time. Typically, when a filtering task cannot fully specify its criteria a priori, and the unknown part can only be determined at runtime, the filtering criteria has to be updated throughout the filtering process. This is often the case with Scada systems. Many network protocols, such as FTP, RTSP (Real Time Streaming Protocol) and SIP (Session Initiation Protocol), establish connections with dynamically-negotiated port numbers. BSD packet filter (BPF) is the most commonly used protocol in Scada systems [12]. It operates on the Libpcap protocol, as described in figure 4.

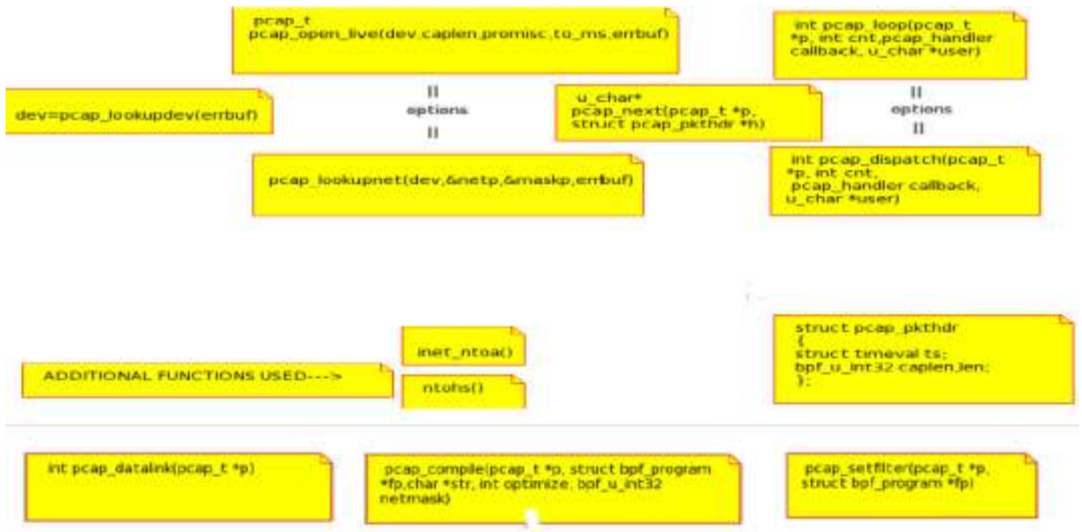
pcap_t
pcap_open_live(dev,caplen,promisc,to_ms,errbuf)
int pcap_loop(pcap_t *p, int cnt,pcap_handler callback, u_char *user)
dev=pcap_lookupdev(errbuf)
||
options
||
u_char*
pcap_next(pcap_t *p, struct pcap_pkthdr *h)
||
options
||
pcap_lookupnet(dev,&netp,&maskp,errbuf)
int pcap_dispatch(pcap_t *p, int cnt, pcap_handler callback, u_char *user)
inet_ntoa()
ADDITIONAL FUNCTIONS USED--->
ntohs()
struct pcap_pkthdr
{
struct timeval ts;
bpf_u_int32 caplen,len;
};
int pcap_datalink(pcap_t *p)
pcap_compile(pcap_t *p, struct bpf_program *fp,char *str, int optimize, bpf_u_int32 netmask)
pcap_setfilter(pcap_t *p, struct bpf_program *fp)

Figure 4. Summary of different function calls that characterize Libpcap library

We identity various problems associated with the Libpcap protocol. Firstly, a filter update in BPF must undergo 3 pre-processing phases: compilation, user–kernel copying, and security checking. In the compilation phase, the filtering criteria specified by the human-oriented pcap filter language are translated and optimized into the machine-oriented BPF filter program. In the user–kernel copying phase, the compiled filter program is copied into the kernel. Finally, in the security checking phase, the kernel-resident BPF instruction interpreter examines the copied filter program for potentially dangerous operations such as backward branches, ensuring that user level optimizer errors cannot trigger kernel misbehavior (for example, infinite loops). Consequently, the whole process of a filter update in BPF induces prolonged latency. As a result, in high speed Scada networks, hundreds or even thousands of packets of interest might be missed by BPF during each filter update, effectively leaving a window of blindness. And for the protocol, if this happens to be a session initiation, the problem is even greater.

The second problem is that BPF uses a RISC (reduced instruction set) like instruction set for a low-level register machine abstraction. Thus, each pcap language primitive is translated into an instruction block that comprises a variable number of simple instructions. Now, when we change a primitive in a filter, it often alters the size of the corresponding instruction block. So firstly, we need to modify code offset related instructions (for example, conditional branch) throughout the entire compiled filter to accommodate this change. Then only, recompilation is possible. The third problem is that the RISC like ISA (instruction set architecture) in BPF induces high instruction interpretation overhead. Interpretation overhead refers to the operations an interpreter must perform before executing an actual filter instruction, such as program counter maintenance, instruction loading, operation decoding, and so forth. These operations are unproductive towards evaluating filter criteria, but cannot be omitted. So, we can very well point out that in addition to filter update latency, the filter execution efficiency of BPF also has a scope of improvement. Recent packet filters such as xPF (x Packet Filter) and FFPF (Fairly Fast packet Filter) move more packet processing capabilities from userspace into the kernel, which reduces context switches and improves overall performance. But, they use kernel space library functions and precompiled binaries, which increases programming complexity. Therefore, we need to devise an optimized mechanism for dynamic packet filtering, a simpler computational model with fewer instructions, the main aim being achieving low filter update latency by avoiding filter recompilation.

Proposed Solution: In the new design approach, we basically need to cluster a series of instructions connected by logic AND. When a packet arrives, the instructions of this group are evaluated one by one. If all evaluation results are true, the packet is accepted and copied to userspace. Otherwise, if any evaluation result is false, the packet fails in the current group, and will be tested by remaining groups. It will be dropped if it fails all groups. Groups are thus effectively independent and are combined by logic OR. In subsequent changes, if a new group is related to one of the existing groups, the application can duplicate an existing group and modify the copy. This has two benefits: (i) Only the difference between the old and new control flows needs to be updated, and thus time is saved. (ii) Secondly, the parent child group relationship can be stored in buffer by the filtering engine, and can be used to optimize filter execution later. Two design choices are proposed to enable in place filter modification [22] [23].

1. Fixing instruction length and Removing filter optimization: By fixing the instruction length, we can avoid the need to shift instructions and change offsets on instruction replacement. By removing filter optimization, not only do we save precious time during a filter update, but also preserve some kind of one-to-one mapping between filtering primitives and filter program instructions. As a result, updates to a filter can be directly applied to the affected instructions without altering filter program structure. This feature further helps to optimize filter update by reducing unnecessary user-kernel data copying. Only the updated part of a filter criterion is copied from userspace to the kernel.

2. Increasing filter execution efficiency: We can achieve this goal by exploring the following two optimizations: Firstly, SIMD (single instruction multiple data) expansion to the instruction set. SIMD allows an interpreter to perform a single instruction interpretation and apply the same operation on many sets of data, thereby significantly reducing the cost of instruction interpretation. Traditionally, SIMD has been widely used in contemporary high performance processors, such as Intel Pentium series and IBM Power series processors. Secondly, to make up for the possible performance loss, hierarchical execution optimization can be introduced. This optimization draws its motivation from the observation that during a dynamic filtering process, the newly added primitives are often related to some existing ones. For example, the new primitives quite often monitor the same host but on different ports or capture the same protocol traffic but for different hosts. Therefore, the existing primitives can be viewed as the parent of the new primitives. We can use this hierarchical relationship among primitives to avoid redundant instruction executions. Also, the execution path of a filter program is determined by a boolean evaluation. So, no storage or branch instruction offsets need to be taken care of.

Thus, in conclusion, we can say that to achieve compilation free update, filtering criteria must map directly onto interpreter instructions. This makes a high-level, CISC-like instruction set architecture a natural choice. In addition to saving compilation time in Scada systems that operate with millions of instructions, the CISC-like instruction set also opens a door for performance optimization as we saw above. A complex instruction is able to accomplish the same task as several simple BPF instructions, thereby reducing instruction interpretation overhead.

5. Generalized solution for Hardware based Dynamic Filtering

Packet sniffers are faced with problems such as high packet loss and cpu utilization due to the amount of traffic to be analyzed. Overall system performance can be improved only if both the kernel and applications do not waste several cpu cycles just for pushing unnecessary packets to user space that will be later discarded. BPF includes a packet filtering machine able to execute programs. Each program is an array of instructions that sequentially execute some actions on a pseudo-machine state. In BPF, checking whether a given packet is Tcp takes 8 instructions. A slightly more complex check such as verifying if a packet is Http, takes more than double the number of instructions. Hardware-based packet filters such as those based on Field programmable gate arrays (FPGA) also have certain limitations. They are limited by the space available on the silicon/RAM used for storing filters. Adding and removing filters can require a general reconfiguration of the hardware (for example, when FPGA's are used). In some case this can take up to a minute, which is not compatible with most traffic monitoring applications in Scada networks that require to operate continuously. Moreover, filters are often not able to detect mixed encapsulation and negative filtering (for example, not <expression>, not (tcp and port 80)) is often not supported [20].

Therefore, our objective for Scada systems is to implement a pre-BPF filtering layer designed on the requirements of emerging traffic monitoring applications. Basically, we aim to reducing the number of packets that are forwarded to the application to be only discarded later on. Dynamic filtering can be implemented directly into the network device driver because this is the earliest place on the system where a packet pops up. This means that if packets are dropped on this component due to filtering, all components on top of the device driver (kernel and user space applications) will benefit from it. Instead, if dynamic filtering is implemented on top of the device driver or, even worse, in user-space, the amount of work needed to move a packet from the device to the filtering component will be wasted for those packets that do not match any filter. From the user’s point of view, this kind of dynamic filtering will be transparent to existing applications. Therefore, no modification will be necessary in order to take advantage of it. Thus, any proposed solution should guarantee constant memory consumption, regardless of the number of filters.

Proposed Solution: It involves application of Bloom filters [13]. It is a probabilistic data structure that is used to test whether an element is a member of a set. False positives are possible, but false negatives are not. The filter is a bit array of 'm' bits, all set to 0. There must also be 'k' different hash functions defined, each of which maps or hashes some set element to one of the 'm' array positions with a uniform random distribution. A false positive error, commonly called a false alarm is a result that indicates a given condition has been fulfilled, when it actually has not been fulfilled. A false negative error is where a test result indicates that a condition failed, while it actually was successful. To add an element, we feed it to each of the 'k' hash functions to get 'k' array positions. Now, we set the bits at all these positions to 1. To query for an element (test whether it is in the set), we feed it to each of the 'k' hash functions to get 'k' array positions. If any of the bits at these positions are 0, the element is definitely not in the set: if it were, then all the bits would have been set to 1 when it was inserted. If all are 1, then either the element is in the set, or the bits have by chance been set to 1 during the insertion of other elements, resulting in a false positive. Insertion and Searching in a bloom filter takes O(1) time. However, deletion is not possible because each bit may have been set to one by several hashes. Hence, if during removal it is set to zero this will break the logic of disallowing false negatives. For this reason if removal is necessary, a counting bloom is used, where each bloom array element is not a bit but a counter that is incremented/decremented every time the bit is set/reset. It's better to use these filters in hardware, as making them software-based is costly in terms of kernel memory usage (continuous allocation) and speed.

To summarize, because bloom filters are space efficient, do not present false negatives and allow elements to be dynamically added from the filter set, they are a good choice for use in Scada systems. For our packet filtering design, it is better to keep a smaller number of hash functions (k). If the bloom size is large and the hash function is good, this would not lead to many false positives. It will also provide a good performance as lesser hashes need to be computed in the worst case, whenever a packet is checked for inclusion in the filter set. Moreover, hash collisions (two different filters producing the same hash result), with a large bloom array, are pretty rare. Whenever a new element is added to the filter set and there is a case of collision, a list of collisions can be maintained. The counter for the collision list will be incremented/decremented based on adding/removing the filter. The only disadvantage of using bloom filters is that this implementation will be NIC (network interface card) dependent.

Bloom filters can be easily implemented using the network API (application programming interface). In Linux operating system, packets are fetched from network adapters using NAPI, a network API that implements efficient packet polling. This means that whenever there is an incoming packet just received by the adapter and ready to be handled by the driver, NAPI does the job by a function call (netif_rx_schedule) that arranges and polls packets out of the adapter. NAPI is based on interrupts. An interrupt is generated as soon as the adapter receives an incoming packet. At this point NAPI takes over the control, disables NIC interrupts and starts polling packets as long as there are packets to receive.

When no more packets are available, interrupts are restored. This implies basically allowing drivers to run with (some) interrupts disabled during times of high traffic to decrease the system load. NAPI also has provisions for packet throttling. When the system must drop packets, it's better if those packets are disposed off, before much effort goes into processing them. NAPI drivers can often cause packets to be dropped in the network adaptor itself, before the kernel sees them at all.

Alternatively, we can use a trie based implementation too [14]. A trie is an ordered tree data structure that is used to store an associative array with help of keys. Unlike a binary search tree, no node in the tree stores the key associated with that node; instead, its position in the tree defines the key with which it is associated. Bitwise tries are much the same as a normal character based trie except that individual bits are used to traverse. Implementations can use a special cpu instruction to very quickly find the first set bit in a fixed length key.

For example, GCC's __builtin_clz (unsigned int x) function, which returns the number of leading 0-bits in 'x', starting at the most significant bit position. This value can then be used to index a 32 or 64 entry table which points to the first item in the bitwise trie with that number of leading zero bits. The search then proceeds by testing each subsequent bit in the key and choosing child(0) or child(1) appropriately until the item is found. Although this process may seem slow, it is very cache-local and highly parallelizable due to the lack of register dependencies. Therefore, it has excellent performance on modern out of order execution cpu's, widely used in Scada systems.

Such data structures that provide constant lookup time with minimal memory utilization can give a significant performance boost, since access to cpu cache is considerably faster than access to RAM. A red-black tree performs much better theoretically. But, it is highly cache-unfriendly and causes multiple pipeline stalls on modern cpu's which makes that algorithm bound by memory latency rather than cpu speed. Rebalancing the trees requires touching more adjacent nodes to rearrange and recolor them, which lowers performance. In comparison, a bitwise trie rarely accesses memory and when it does it does so only to read, thus avoiding cache coherency overhead in symmetric multiprocessing. Hence it is the algorithm of choice, for code which does a lot of insertions and deletions. For example, in our case of packet filtering, where packets are accepted/rejected/queued/forwarded based on dynamic rules insertion and deletion. We propose the inclusion of these data structures in form of following two options:

1. The bloom or trie based filtering can be implemented into the polling NAPI function. Only those packets that satisfy these filters will be returned. Those having no corresponding filtering rule will automatically be dropped at this stage and will never be handled by NAPI. This option will just be an enhancement over the existing NIC driver.

2. The bloom or trie based filtering can be implemented at a lower level than NAPI polling function netif_rx_schedule(). Packets will be polled as soon as they become available. They will then be filtered with the above described two approaches, and if they pass this stage, they will be pushed up the network stack by calling the netif_rx_schedule( ) function. The clear cut advantage of this approach is that packet latency will be decreased, as the NIC will push packets only when necessary. So, we don't have to worry about interrupts from NIC to the kernel now.

Overall, the advantage of the above proposed solutions is that packets do not enter at all in the protocol stack. So, there is no need to allocate memory (in struct skbuff or socket buffer) to store the packet, that will then be deallocated later when the packet is dropped. On top of that, reducing the number of memory or deallocations is a significant performance improvement, as this has to be done for each incoming packet on the network.

6. Analysis of High Speed Packet Capturing

In this section, we aim to analyze packet traces at high speed for Scada systems. As correctly pointed by Cascallana et al. [24], the biggest problem associated with packet capture is reducing the number of dropped packets. In networks where traffic is relatively low, packet loss may be trivial or even nonexistent, but when capturing packets in high speed Scada network environments, packet loss can be substantial. To understand packet loss we first need to understand what happens to a packet when it is received by a machine. Under normal conditions, when an NIC sees a packet, it determines if the destination MAC address in the packet matches its own MAC address or the link layer broadcast address. If it does not match, the packet is discarded. When a packet that matches is received by the machine's NIC, the packet is copied into a buffer and the NIC informs the kernel that it has received a packet that needs to be processed. The packet is then passed from the NIC to the kernel. The kernel processes the packet by performing error checking and striping off layers of the packet as it passes through the kernel. The packet is then passed to the appropriate socket based on information from the packet header and queued for use by a user application such as Ssh or Apache. The application performs a system call to get the packet and then clears the queue.

As the packet propagates from NIC to the kernel and then to the userspace application, it creates some overhead. Under heavy traffic conditions the percentage of the captured packets over the total number can decrease. In a situation where a user wants to capture all packets, regardless of its destination or correctness, the NIC needs to be placed in promiscuous mode. In essence, this tells the NIC to accept all packets whether they are destined for that specific NIC or not. Again, the packet is copied into a buffer and, the NIC informs the kernel that it has received a packet that needs to be processed. Once again, the packet is then passed from the NIC to the kernel. At this point something different needs to happen. Since the goal is to capture all packets as they were seen on the network, it is not desirable for the kernel to discard or strip off parts of the packets during its processing, so a copy of the packet is made. One copy is sent to the upper-levels of the kernel, the same as before, where most are discarded because they are not destined for this machine. The other unaltered packet is passed to the socket of a user packet capture application and queued for use.

In most instances, this socket is created by the packet capture library Libpcap. Libpcap provides a common API for packet capture. Libpcap is responsible for performing the system call to get the packet and clearing the queue. According to Johnson et al. [15], the linux kernel receives each frame and subjects it to a four-step process:

1. Hardware Reception: The network interface card (NIC) receives the frame on the wire. Depending on its driver configuration, the NIC transfers the frame either to an internal hardware buffer memory or to a specified ring buffer.
2. Hard IRQ: The NIC asserts the presence of a net frame by interrupting the cpu. This causes the NIC driver to acknowledge the interrupt and schedule the soft IRQ operation.
3. Soft IRQ: This stage implements the actual frame-receiving process, and is run in softirq context. This means that the stage pre-empts all applications running on the specified cpu. The kernel actually removes the frame from the NIC hardware buffers and processes it through the network stack. From there, the frame is forwarded, discarded, or passed to a target listening socket. When passed to a socket, the frame is appended to the application that owns the socket. This process is done iteratively until the NIC hardware buffer runs out of frames, or until the device weight (dev_weight).
4. Application receive: the application receives the frame and dequeues it from any owned sockets via the standard posix calls (read, recv, recvfrom). At this point, data received over the network no longer exists on the network stack

   In Scada systems, the most common reasons for packet or frame loss could be queue overrun and the queue being full. The difference is nature of these two problems motivate the development of interesting and partly intersecting solutions, which we describe as follows:

1. The most common reason for frame loss is a queue overrun. The kernel sets a limit to the length of a queue, and in some cases the queue fills faster than it drains. When this occurs for too long, frames start to get dropped. Packet loss can occur in a couple of places. The NIC fills its hardware buffer with frames; the buffer is then drained by the softirq, which the NIC asserts via an interrupt. If the buffer used by the NIC to hold packets before being passed to the kernel is full, then the NIC will drop the new packet. This can occur if the NIC is receiving packets at an extremely fast rate or if the kernel is too busy performing other tasks to receive the packet. We can interrogate the status of this queue by #ethtool -S ethX

Proposed Solution: Firstly, we can slow down input traffic by filtering frames or preemptively dropping them, and lowering broadcast traffic. Secondly, we can increase the queue length. This involves increasing the number of buffers in a specified queue to whatever maximum the driver will allow. To do so, we can edit the rx/tx ring parameters of ethX using # ethtool --set-ring ethX [rx N] [tx N]. Thirdly, we can increasing the rate at which a queue is drained. To do this, we can adjust the NIC's device weight accordingly. This attribute refers to the maximum number of frames that the NIC can receive, before the softirq reschedules itself. It is controlled by the /proc/sys/net/core/dev_weight variable. It is the maximum number of packets that kernel can handle on a NAPI interrupt per cpu. One possible disadvantage with the proposed solution is that if we increase the rate at which a queue is drained, it indirectly implies increasing the number of frames that can be received from a NIC in one iteration. So, extra cpu cycles will be used, during which no applications can be scheduled on that cpu.

2. Packets can also be lost if the socket queue is full when the kernel attempts to pass the copy of the packet to libpcap's socket. The socket queue is filled by the network stack from the softirq context. Applications then drain the queues of their corresponding sockets via calls to read( ), recvfrom( ) system calls. This can occur if the application using libpcap is unable to process packets fast enough, due to lack of resources.

Proposed Solution: Firstly, we can optimize an application to perform calls more frequently. This involves modifying or reconfiguring the network application to perform more frequent posix calls (such as recv, read). In turn, this allows an application to drain the queue faster. Secondly, we can increase queue depth by changing a kernel parameter that controls the default size of receive buffers used by sockets, that is, #sysctl -w net.core.rmem_default=N. We can view this changed value by accessing /proc/sys/net/core/rmem_default on Unix machines. Also, value of rmem_default should be no greater than rmem_max (/proc/sys/net/core/rmem_max)

In both cases, system utilization can play a big part in packet loss. Placing the NIC in promiscuous mode to allow packet capture greatly increases the number of packets the kernel must handle, and utilizes a lot more cpu cycles than would be normally required for network operations. This increases load, which in turn can reduce the ability of the system to capture packets. The size of the frame does play a significant factor, as the smaller the packet size, the higher the negative impact in the packet capture percentage. The reason for this is that, for same throughput, the amount of smaller packets is greater than for bigger packet sizes. This results in more need for processing power. One of the better solutions is to enhance or configure the application to drain data from the kernel more quickly, even if it implies queuing the data in application space. This will allow the data be stored more flexibly, since it can be swapped out and paged back in, as needed.

7. Future Research Directions

Libpcap is an open source library for packet capturing and uses by default Pf_Packet protocol in order to transfer the packets from the driver to the userspace. It provides an API for the programmer to select the capturing interface (device) and gives the ability to compile Linux Packet Filters (Lpf) into the kernel for selective packet capturing based on the 5 tuple (protocol, source/destination IP address and source/destination port). However, as a future research direction for Scada systems, Pf_Ring [16] is a promising replacement for Pf_Packet that provides a kernel module and modifications to libpcap. This allows an application to create a socket designed for improved packet capture, through the use of a ring buffer and shared memory. Pf_Ring uses memory mapping instead of processing expensive buffer copies from kernel space to userspace and uses ring buffers to make this transportation more efficient. It not only provides a way to prevent packets from being passed to the upper-levels of the kernel, but also inspects at a much lower level than traditional packet sniffers/engines, therefore reducing resource cost and increasing overall efficiency. This smarter alternative reduces the amount of work, the kernel needs to perform for each packet, which in turn improves packet capture performance.

Pf_Ring allows packets on a single interface to be segmented across multiple threads/cores, allowing for more efficient packet processing. For the end user, this can be done by enabling Receive Side Scaling (RSS) [17] on Intel network adaptors. There is no delay due to context switches or spinlock. Moreover, no semaphores need to be used (as in case of sequential processing from the queue). Generally, protocol processing done in the NAPI context for received packets is serialized per device queue and becomes a bottleneck under high packet load. So, Pf_Ring provides a parallelized mechanism to distribute the load of received packet processing across multiple cpu's. However, we feel that an alternative and easier implementation can be done by:

1. Enabling a single NIC rx queue to have its 'receive softirq workload' distributed among several cpu's: It's basically selecting the cpu to perform protocol processing above the interrupt handler. This will inturn help prevent network traffic from being bottlenecked on a single NIC hardware queue. We can do so by specifying the target cpu names in # /sys/class/net/ethX/queues/rx-N/rps_cpu on Unix machines and then replacing ethX with the NIC's corresponding device name (for example, eth1, eth2) and rx-N with the specified NIC receive queue. This will allow the specified cpu's in the file to process data from queue rx-N on ethX

2. Changing the file # /sys/class/net/ethX/queues/rx-N/rps_flow_cnt on Unix machines: This is the file that controls the maximum number of sockets/flows that the kernel can steer for a specified receive queue (rx-N) on a NIC (ethX). The main point here is to increase data cache hitrate, by steering kernel processing of packets to the cpu, where the application thread (which is consuming packets) is running.

However, there are three important limitations in using the above methodology. Firstly, Pf_Ring polls packets from NICs by means of Linux NAPI. NAPI copies packets from the NIC to the Pf_Ring circular buffer, and then the user application reads packets from ring. So, there are two pollers, both the application and NAPI. This results in usage of double cpu cycles for these purposes, which is an overhead. Secondly, Receive Side Scaling (RSS) has an unbalanced network data flow on multiple processor cores. Manipulating this inbuilt balance function of Intel network adaptors is another problem. For Direct NIC Access (DNA), packets are generally processed in sequence (in FIFO) whereas applications sometimes may need to store packets and process them out of sequence. In case of fragmented packets fir a TCP/IP protocol, a given packet must be rebuilt with all fragments prior to process it, considering fragment offset and length.

A fascinating enhancement would include using the concept of Direct Memory Access (Dma). What this means is that NIC memory and registers will be directly mapped to any user application. Instead of using NAPI polling, it's better if we use the NIC network process unit's polling only. Afterwards, we can use Direct memory access (Dma) to transfer filtered packets directly to user application. So, the cpu cycles will only be used for consuming the packets rather than performing any unwanted copying. There is no need for continuous intervention by the processor. Bus Arbitration (centralized/distributed) may be followed to decide which device can initiate data transfers in the bus first, in case of conflict between the processor and Dma controller. Some other interesting future work will involve:

1. Implementation of more advanced security modules for FlexiDnp3 protocol that address one-many design topology.
2. Improving the Pf_Ring patch for packet filtering kernel libraries.
3. Integrating more network performance tuning concepts to optimize protocol working for larger Scada networks. In real time Scada networking, where lots of log files and binaries are generated, it is necessary to keep record of original file system attributes. Using primary operating system tools such as tcpdump, md5sum, netstat, nmap etc, we can have control over the Scada network infrastructure. For example, the md5sum tool cryptographically hashes the contents of any file using a variation of Md5 algorithm. The result is a string of characters that can be used as a fingerprint for the file. Two different files hashing to the same value is infeasible. So, if we run md5sum, fingerprints of important Scada system binaries can be taken, and they can be compared to a baseline record of their original values.
4. Using linux network services security like public key cryptography, server monitoring and intrusion detection systems [21] to address the threats posed to Scada systems at user application layer, in addition to hardening only at the protocol level.
5. Performing time series analysis of DnpSec and FlexiDnp3 Protocols in case of bigger Scada networks, and understanding response actions based on it.

8. Conclusion

In this work, we analyzed various packet filtering libraries and their inadequacies. Firstly, we dived into the data structures used in libipq and libpcap libraries, and then proposed solutions that could be used to overcome potential flaws, which can get aggravated in large scale Scada networks. After suggesting new design solutions pivoted on improving efficiency at the architectural level, we moved on to describe a generalized solution for hardware based dynamic filtering. Using memory efficient data structures such as bloom filters and trie, we illustrated how packet filtering mechanisms could be modified for better Scada security. We also pointed out different hardware and software levels at which such methods could be deployed, highlighting pros and cons for each strategy. Finally, we wrapped up using novel performance tuning approaches to provide fundamentally sound and optimal solutions to side effects such as packet loss, which can cause security breaches in Scada systems. In the end, we described future research directions for high speed packet capturing in parallel, which is consequential for increasing performance and providing strong security measures in Scada networks.

Acknowledgment

This work was originally started by Tanmay Sinha at Center for Development of Advanced Computing (CDAC), Mumbai, India and later continued by both the authors. The authors gratefully acknowledge the help and supervision of Mr. Ashwin Nivangune, Mr. Shashi Bhushan Prabhakar, Mr. Paritosh Tripathi (Technical officers and Staff scientists, CDAC Mumbai) and Dr. Zia Saquib (Executive Director, CDAC Mumbai).

References

[1]. Boyer, S. A. (2009). Scada: supervisory control and data acquisition. International Society of Automation.
[2]. Cleveland, F. (2006, May). Iec TC57 Security Standards for the Power System's Information Infrastructure-Beyond Simple Encryption. In Transmission and Distribution Conference and Exhibition, 2005/2006 IEEE PES (pp. 1079-1087). IEEE.
[3]. Curtis, K. (2005). A Dnp3 protocol primer. Dnp3 User Group.
[4]. Clarke, G. R., Reynders, D., & Wright, E. (2004). Practical Modern Scada Protocols: Dnp3, Iec 60870.5 and Related Systems. Newnes.
[5]. East, S., Butts, J., Papa, M., & Shenoi, S. (2009). A Taxonomy of Attacks on the Dnp3 Protocol. In Critical Infrastructure Protection III (pp. 67-81). Springer Berlin Heidelberg.

[6]. Majdalawieh, M., Parisi-Presicce, F., & Wijesekera, D. (2006). DnpSec: Distributed network protocol version 3 (Dnp3) security framework. In Advances in Computer, Information, and Systems Sciences, and Engineering (pp. 227-234). Springer Netherlands.
[7]. Bagaria, S., Prabhakar, S. B., & Saquib, Z. (2011, December). Flexi-Dnp3: Flexible distributed network protocol version 3 (Dnp3) for Scada security. InRecent Trends in Information Systems (ReTIS), 2011 International Conference on (pp. 293-296). IEEE.
[8]. Jain, P., & Tripathi, P. (2013). Scada security: a review and enhancement for Dnp3 based systems. CSI Transactions on ICT, 1-8. Jain, P., & Tripathi, P. (2013). Scada security: a review and enhancement for Dnp3 based systems. CSI Transactions on ICT, 1-8.
[9]. Team, N. C. (2006). Libipq-Iptables Userspace Packet Queuing Library.
[10]. Jacobson, V., & McCanne, S. (2009). libpcap: Packet capture library. Lawrence Berkeley Laboratory, Berkeley, CA.
[11]. Schneider, F. B. (2003). Least privilege and more [computer security]. Security & Privacy, IEEE, 1 (5), 55-59.
[12]. McCanne, S., & Jacobson, V. (1993, January). The BSD packet filter: A new architecture for user-level packet capture. In Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings (pp. 2-2). USENIX Association.
[13]. Broder, A., & Mitzenmacher, M. (2004). Network applications of bloom filters: A survey. Internet Mathematics, 1 (4), 485-509.
[14]. Willard, D. E. (1984). New trie data structures which support very fast search operations. Journal of Computer and System Sciences, 28 (3), 379-394.
[15]. Johnson, S. K., Huizenga, G., & Pulavarty, B. (Eds.). (2005). Performance tuning for Linux servers. IBM.
[16]. Deri, L., & PF RING, M. (2011). URL http://www. ntop. org. Pf_Ring. html.
[17]. Veal, B. E., & Foong, A. (2007). U.S. Patent Application 11/771,250.
[18]. Tsang, P. P., & Smith, S. W. (2008, January). YASIR: A low-latency, high-integrity security retrofit for legacy Scada systems. In Proceedings of The Ifip Tc 11 23rd International Information Security Conference (pp. 445-459). Springer US.
[19]. Stevens, W. R. (2004). UNIX network programming (Vol. 1). Addison-Wesley Professional.
[20]. Gouda, M. G. (1998). Elements of network protocol design. John Wiley.
[21]. Yang, D., Usynin, A., & Hines, J. W. (2006, November). Anomaly-based intrusion detection for Scada systems. In 5th Intl. Topical Meeting on Nuclear Plant Instrumentation, Control and Human Machine Interface Technologies (NPIC&HmiT 05) (pp. 12-16).
[22]. Pande, B., Gupta, D., Sanghi, D., & Jain, S. K. (2005, July). The Network Monitoring Tool—PickPacket. In Information Technology and Applications, 2005. ICITA 2005. Third International Conference on (Vol. 2, pp. 191-196). IEEE.
[23]. Cline, W. W. (2006). A prototype for in situ packet filtering (Doctoral dissertation, College of William and Mary).
[24]. Cascallana, G. A., & Lizarrondo, E. M. (2006, September). Collecting packet traces at high speed. In Proc. of Workshop on Monitoring, Attack Detection and Mitigation (MonAM) 2006.